A GPU-Enabled Solver for Time-Constrained Linear Sum Assignment Problems

Roberto Roverso *†, Amgad Naiem*‡, Mohammed El-Beltagy*‡, Sameh El-Ansary*§ and Seif Haridi†
*Peerialism Inc., Sweden
†KTH-Royal Institute of Technology, Sweden
‡Cairo University, Egypt
‡Nile University, Egypt
{roberto,amgad,mohammed,sameh}@peerialism.com haridi@kth.se

***Abstract*—This paper deals with solving large instances of the Linear Sum Assignment Problems (LSAPs) under real-time constraints, using Graphical Processing Units (GPUs). The motivating scenario is an industrial application for P2P live streaming that is moderated by a central tracker that is periodically solving LSAP instances to optimize the connectivity of thousands of peers. However, our findings are generic enough to be applied in other contexts. Our main contribution is a parallel version of a heuristic algorithm called Deep Greedy Switching (DGS) on GPUs using the CUDA programming language. DGS sacrifices absolute optimality in favor of a substantial speedup in comparison to classical LSAP solvers like the Hungarian and auctioning methods. We show the modifications needed to parallelize the DGS algorithm and the performance gains of our approach compared to a sequential CPU-based implementation of DGS and a mixed CPU/GPU–based implementation of it.**

I. Introduction

In order to deal with hard combinatorial optimization problems in a time-constrained environments where the time to compute a solution is bounded by the characteristics of the system, it is often necessary to sacrifice optimality in order to meet the imposed deadlines.

In our experience, we have dealt with a large scale peer-to-peer live-streaming platform where the task of assigning n senders to n receivers is carried out by a centralized optimization engine. The problem of assigning peers to one-another is modeled as a *linear sum assignment problems* (LSAP). However, due to the scale of the p2p system, the computational overhead of minimizing the cost of assigning n jobs (receivers) to n agents (senders) is usually quite high because of the size of the problem i.e. the number peers in the system. We have seen our implementation of classical LSAP solvers take several hours to provide an optimal solution to a problem of this magnitude.

In the context of our live streaming application we could afford only a few seconds for the optimization process to terminate. It was also of great importance for us not to sacrifice optimality too much in the pursuit of a viable and timely solution to our problem.

We therefore resorted to design a fast heuristic near-optimal solver for LSAP which is also amenable to parallelization in such a way that can make use of the massive computational potential of modern Graphic Processing Units.

After a number of iterations and structured evaluation of different ideas for a heuristic optimizer, we found a simple and effective heuristic which we called Deep Greedy Switching [1] (DGS). It was shown to work extremely well on the instances of LSAP we were interested in, and we never observed it deviate from the optimal solution by more than 0.6%, (c.f. [1, p. 5]). Seeing that DGS has great parallelization potential, we modified and adapted it to be run on any parallel architecture and consequently also on GPUs.

In this work, we chose the CUDA [2] as a GPU programming language to implement the solver. CUDA is a sufficiently general C-like language which allows for execution of any kind of user-defined algorithms on the highly parallel architecture of NVIDIA GPUs.

GPU programming has become increasingly popular in the scientific community during the last few years. However, the task of developing whatsoever mathematical process in a GPU-specific language still involves a fair amount of effort in understanding the hardware architecture of the target platform. CUDA is no exception, one must still understand the basics of the functioning of NVIDIA GPUs and be acquainted with a number of best practices to be able to achieve best performance. Considered that, in this paper we'll provide a short introduction to CUDA in Section II, for a better understanding of its advantages, best practices and limitations, which will later justify our design choices in Section V. We will then describe the DGS heuristic in Section III and the result of adapting the algorithm to be run on GPUs compared to other implementations of the same DGS in Section VI.

II. GPUs and The CUDA language

Graphical Processing Units are mainly accelerators for graphical applications, such as games and 3D modeling software, which make use of the OpenGL and DirectX programming interfaces. Given the parallel nature of those applications, GPUs have hence been architected as massive parallel machines. In the last years however, GPUs have stopped being exclusively fixed-function devices to become flexible parallel processors accessible through programming languages [2][3]. In fact, modern GPUs as NVIDIA Tesla [4] and GTX are fundamentally fully programmable many-core chips, each one of them having a large number of parallel processors. Multicore

chips are called Streaming Multiprocessors (SMs) and their number can vary from one, for low-end GPUs, to as many as thirty. A single SM contains in turn 8 scalar Scalar Processors (SPs), each equipped with a set of registers, and 16KB on-chip memory called Shared Memory. The latter memory has very low access latency, high bandwidth and, if used in the right fashion, can provide substantial performance gains compared to off-chip memory, called Global Memory. Global Memory is usually of the DDR3 or DDR5 type and it is fully addressable by each SP. Off-chip memory is much slower compared to Shared Memory but much more abundant.

We chose CUDA as GPU Computing language for implementing our solver because it best accomplishes a trade-off between ease-of-use and required knowledge of the hardware platform's architecture. Other GPU specific languages, such as AMD's Stream [5] and Kronos' OpenCL standard [3] look promising but fall short of CUDA either for the lack of support and documentation or for the quality of the development platform in terms of stability of the provided tools, such as compilers and debuggers.

To ease the task of implementing parallel algorithms, CUDA provides a sufficient degree of abstraction from the GPU architecture. Yet, one must still understand the basics of the functioning of NVIDIA GPUs to be able to fully utilize the power of the language. The CUDA programming imposes that the application to be organized in a *sequential part* running on a *host*, usually the machine's CPU, and parallel parts called *kernels* that execute code on a parallel *device*, i.e. the GPU(s). Kernels are blocks of instructions which are executed across a number of parallel threads. Those threads are logically organized by CUDA in a grid whose sub-parts are the *thread blocks*. A thread block is a set of threads which can synchronize themselves across the thread block exclusively using barrier synchronization. Every block can access an amount of Shared Memory which is exclusive for its group of threads. The number of blocks and the number of threads for each block are specified when launching the kernel. The blocks are therefore a way for CUDA to abstract the physical architecture of Scalar Multiprocessors and Processor away from the programmer. Management of Global and Shared Memory must be enforced explicitly by the programmer through primitives provided by CUDA. Although Global memory is sufficient to run any CUDA program, it is advisable to use Shared Memory in order to obtain efficient cooperation and communication between threads in a block. It is particularly advantageous to let threads in a block load data from global memory to shared on-chip memory, execute the the kernel instructions and later copy the result back in global memory.

III. DGS Heuristic

In this paper we are interested in considering the classical assignment problem of finding the optimal assignment of n agents to n jobs, where there is a certain benefit a_{ij} given when assigning agent i to job j. The optimal assignment of agents to jobs is the one that yields the maximum total benefit given that each agent can only be assigned to only one job and same for each job. The assignment problem is formally described as an *Integer Linear Programming Problem* as follows

$$\max \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ij} x_{ij}$$

$$\sum_{i=1}^{n} x_{ij} = 1 \qquad \forall j \in \{1 \dots n\}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \forall i \in \{1 \dots n\}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \in \{1 \dots n\}$$

As the LSAP is common to many applications and domains, a number of algorithms has been developed specifically to solve it, exploiting some of its characteristics. The most popular algorithms for solving the LSAP are the Hungarian method [6] and the auction algorithm [7]. The auction algorithm has been shown to be very effective in practice, for most instances of the assignment problem. The algorithm works like an auction where there is a price for each job that is set to zero at the beginning of the algorithm and all agents starts unassigned as well. At each iteration, unassigned agents bid simultaneously on their "best" jobs which causes the jobs' prices to rise accordingly. The algorithm keeps iterating until all agents are assigned.

We found that the auction algorithm falls short of our needs as we have a dynamic systems that deal with large instances of the assignment problem where a solution the needs to be found in limited time for it to be of practical use. We hence designed a novel heuristic approach called *Deep Greedy Switching (DGS)* [1] to address our challenge. The DGS algorithm, shown in Algorithm 1, starts with a random initial solution, and then keeps moving to better solutions by examining a restricted 2-exchange neighborhood. The DGS approach can be described briefly in the following four steps:

A. Initial Solution

An initial solution can simply be found by getting a random solution, where each agent is assigned to a random job regardless of how good this assignment is, in terms of the benefit added by the a_{ij} value. An alternative way of finding an initial solution is to have each agent find the best possible job from the set of available jobs, and then this job is removed from the set of available jobs. In our experiments we have seen that different initial solutions approaches do not affect the quality of the final solution, nor does it affect the speed.

B. Difference Evaluation

This is where we search for better solutions and it is considered the most important part of the DGS approach and the most expensive in terms of computational time. In this step, starting from a given solution σ, each agent tries to find the best solution from the neighborhood of the solution σ. The neighborhood of σ for each agent are the solutions that involve

the change of the assignment for the agent doing the difference evaluation, noticing that only solutions that are better than the current solution σ are taken into consideration. The difference or the improvement in the objective function between the new solution and the current solution σ is recorded as well. This is called *agent difference evaluation (ADE)*.
The same is done for each job using *job difference evaluation (JDE)*.

C. Sorting Differences

In this step, we sort the solutions in the neighborhood N by the difference in objective function from the current solution σ in descending order such that the best solution is in the beginning.

D. Switching

This is the core of the DGS approach where we go from one solution to a better one. We choose the first solution from the sorted neighborhood N and we replace the current solution with this solution. Of course according to how we search for solutions, the solutions we select from will contain a restricted 2-exchange solutions involving switches between only two agents and two jobs. Hence after we apply the 2-exchange by switching the assignment of these two agents and jobs, we remove any solutions that was added by any of the two agents and two jobs involved in this switch and we also re-evaluate the differences for them. Finally we repeat the switching step until the neighborhood N is empty.

The DGS approach then keeps on repeating the last three steps until there is no change in the solution when doing the repetition.

The algorithm defines $\sigma : J \rightarrow I$, where J is the set of jobs and I is the set of agents, as an assignment mapping such that $\sigma(j) = i$ means that job j is assigned to agent i. Similarly another assignment mapping $\tau : I \rightarrow J$ is for mapping jobs to agent where $\tau(i) = j$ means that agent i is assigned to job j. There is also an assignment mapping function to construct τ from σ defined as $\tau = M(\sigma)$ and the objective function value of an assignment σ is given by $f(\sigma)$.

The algorithm also defines a function called $\textbf{SWITCH}(i, j, \sigma)$ that gives the neighbor of the solution σ where the assignment of agent i is replaced by j or in other words a 2-exchange between i and $\sigma(j)$. Hence the algorithm defines the best switch for agent i and the value of the difference in the objective function for this switch as follows

$$j_i = \underset{j=1,\ldots,n, j\neq\tau(i)}{\arg\max} f(\textbf{SWITCH}(i, j, \sigma)) - f(\sigma),$$

and

$$\overline{\delta}_i = \max_{j=1,\ldots,n, j\neq\tau(i)} f(\textbf{SWITCH}(i, j, \sigma)) - f(\sigma).$$

Same is done for each job but they are named i_j and $\underline{\delta}_j$ respectively. Using these terminology, the algorithm can be formally described as follows

ALGORITHM DGS (σ, f)
repeat
  $\sigma_{start} \leftarrow \sigma$, $\tau = M(\sigma)$, $\overline{\delta} \leftarrow \emptyset$, $\underline{\delta} \leftarrow \emptyset$
  $ADE(i, f, \tau, \sigma, NA, \overline{\delta})$ $\forall i \in I$
  $JDE(j, f, \tau, \sigma, NJ, \underline{\delta})$ $\forall j \in J$
  while $\exists \overline{\delta}_i > 0 \vee \exists \underline{\delta}_j > 0$ **do**
    $i^* \leftarrow \arg\max_{i=1\ldots n} \overline{\delta}_i$, $j^* \leftarrow \arg\max_{j=1\ldots n} \underline{\delta}_j$
    if $\overline{\delta}_{i^*} > \underline{\delta}_{j^*}$ **then**
      $\sigma' \leftarrow \textbf{SWITCH}(i^*, j_{i^*}, \sigma)$, $\tau' \leftarrow M(\sigma')$
      $agents \leftarrow \{i^*, \sigma'(\tau(i^*))\}$,
      $jobs \leftarrow \{\tau(i^*), \tau'(i^*)\}$
    else
      $\sigma' \leftarrow \textbf{SWITCH}(i_{j^*}, j^*, \sigma)$
      $agents \leftarrow \{\sigma(j^*), \sigma'(j^*))\}$,
      $jobs \leftarrow \{j^*, \tau(\sigma'(j^*))\}$
    if $f(\sigma') > f(\sigma)$ **then**
      $\sigma \leftarrow \sigma'$, $\tau = M(\sigma)$
      $ADE(j, f, \tau, \sigma, NA, \overline{\delta})$ $\forall i \in agents$
      $JDE(j, f, \tau, \sigma, NJ, \underline{\delta})$ $\forall j \in jobs$
until $f(\sigma_{start}) = f(\sigma')$;
output σ'

Algorithm 1: DGS algorithm

ALGORITHM ADE $(i, f, \tau, \sigma, NA, \overline{\delta})$
$j \leftarrow \tau(i)$, $\sigma_i^* \leftarrow \sigma$
foreach $j' \in \{J \mid j' \neq j\}$ **do**
  $i' \leftarrow \sigma(j')$
  $\sigma_i' \leftarrow \sigma$, $\sigma_i'(j) = i'$, $\sigma_i'(j') = i$
  if $f(\sigma_i') > f(\sigma_i^*)$ **then**
    $\sigma_i^* \leftarrow \sigma_i'$
if $\sigma_i^* \neq \sigma$ **then**
  $NA_i \leftarrow \sigma_i^*$
  $\overline{\delta_i} \leftarrow f(\sigma_i^*) - f(\sigma)$

Algorithm 2: ADE algorithm

IV. EVALUATION

While explaining the process of realization of the CUDA solver in the next section, we also show results of the impact of the various steps that we went through to implement it and enhance its performance. The experimental setup for the tests consists of a consumer machine with a 2.4Ghz Core 2 Duo processor equipped with 4GB of DDR3 RAM and a NVIDIA GTX 295 graphic card with 1GB of DDR5 on-board memory. The NVIDIA GTX 295 is currently NVIDIA's top-of-the-line consumer video card and boasts a total number of 30 Scalar Multiprocessors and 240 Processors, 8 for each SM, which run at a clock rate of 1.24 GHz. In the experiments, we use a thread block size of $t = 256$ when executing kernels which do not make use of Shared Memory, and $t = 16$ in the case they do.

Concerning the DGS input scenario, we use dense instances of the GEOM type defined by Bus and Tvrdık[8], and generated as follows: first we generate n points randomly in a 2D graph square of dimensions $[0, C] \times [0, C]$, then each a_{ij} value is set as the Euclidean distance between points i and j

from the generated n points. We define the problem size to be equal to the number of agents/jobs. For the sake of simplicity, we use problem sizes which are multiple of the thread block size.

Note that each of the following experiments is the result of averaging a number of runs executed using differently seeded instances of the GEOM input problem.

V. The DGS CUDA Solver

The first prototype of the DGS solver was implemented in the Java language. However, its performance did not meet the demands of our target real-time peer-to-peer system. We therefore ported the same algorithm to pure C language in the hope that we obtain better performance. The outcome of this effort was the first production implementation of the DGS which was sufficiently fast to handle problem sizes of 5000 peers. In order to improve the solver for handling a larger amount of clients, we went through the process of profiling the various parts of the algorithm's implementation. The result of this analysis showed that the Difference Evaluation phase of the algorithm III-B was undoubtedly the most computationally expensive, around 70% of the total computational time needed by the solver. Luckily, all JDE and ADE evaluations for agents and jobs can be done in parallel as they are completely orthogonal and they do not need to be executed in a sequential fashion. Hence, our first action point was therefore to implement a CUDA kernel which would execute the ADE/JDE algorithm on the GPU.

We developed two versions of the JDE/ADE kernel: the first which runs exclusively on the GPU's Global memory and a second which makes use of the GPU's Shared memory to obtain better performance. For ease of exposition we will only discuss ADE going forward. This is without any loss of generality as everything that applies to ADE also applies to JDE, with the proviso the talk of jobs instead of agents.

A. Difference Evaluation on Global Memory

As mentioned earlier, Global memory is fully addressable by any thread running on the GPU and no special operation is needed to access data on it. Therefore, in the first version of the kernel, we decided to simply upload the full $A_{i,j}$ matrix to the GPU memory together with the current agent to job assignments and all the data we needed to run the ADE algorithm on the GPU. Then we let the GPU spawn a thread for each of the agents involved. Consequently, thread ct_i associated with agent i will then execute the ADE algorithm only for agent i by evaluating all possible 2-exchanges. The agent-to-thread allocation on the GPU is trivial and is made by assigning the thread identifier ct_i to agent i.

B. Difference Evaluation on Shared Memory

The second version of the Difference Evaluation kernel makes use of Shared Memory and assigns one thread to every 2-exchange evaluation between agent i and job j. That implies that the number of created threads equals the number of cells of the $A_{i,j}$ matrix. Each thread $t_{i,j}$ then proceeds to load in shared memory the data which is needed for the single evaluation between agent i and job j. Once the 2-exchange evaluation is computed, every thread $ct_{i,j}$ stores the resulting value in a matrix located in global memory in position (i,j). After that, another small kernel is executed which causes a thread for each row i of the resulting matrix to find the best 2-exchange value along that same row for all indexes j. The outcome of this operation represents the best 2-exchange value for agent i. In Figure 1, we compare the results obtained by running the two aforementioned Shared Memory GPU kernel implementations and its Global Memory counterpart against the pure C implementation of the Difference Evaluation for different problem sizes. For evaluation purpose, we used a CUDA-enabled version of the DGS where only the Difference Evaluation part of the algorithm runs on the GPU and can be evaluated separately from the all other parts.

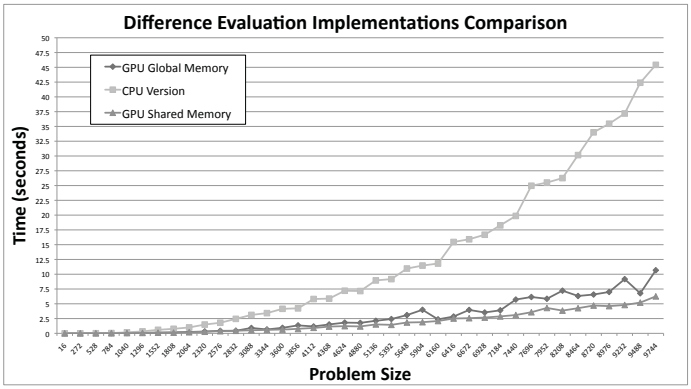

Fig. 1. Computational time comparison between Difference Evaluation's implementations

As we can see, there's a dramatic improvement when passing from the CPU implementation of the difference evaluation to both GPU implementations, of which the Shared Memory version behaves consistently better than the Global Memory one. Furthermore, the trend for increasing problem sizes is linear for both GPU versions of the Difference Evaluation, opposed to the exponential growth of the CPU version curve.

C. Switching

Considering the Switching part of the DGS algorithm described in Subsection III-D we found out that in many cases the computational load necessary to apply the best 2-exchanges is fairly high. Figure 2 shows the relative impact of the Switching part of the algorithm with respect to the total time of execution of the DGS solver for increasing problem sizes using a fixed input scenario for an implementation where only the Difference Evaluation phase is executed on the GPU. As we can see, the load, shown in light grey, can be as prominent as 70% of the total load imposed by the solver. In order to improve performance on this section of the solver, we modified the Switching algorithm so that part of the best 2-exchanges computed in the Difference Evaluation section might be applied concurrently. The modified DGS algorithm is shown in Algorithm 3. In order to execute part of the switches

in parallel, we need to identity which possible exchanges are not conflicting. For that, we designed a function called CC, shown in Algorithm 4, which detects which of the possible 2-exchanges are conflicting. Once the non-conflicted exchanges are determined by CC, we identify the corresponding agents and jobs and we apply their switches in a parallel fashion. After all the aforementioned switches are applied, we proceed to re-evaluate the differences for the agents and jobs whose possible 2-exchanges were not-conflicting, for there might be a possible improvements for those when evaluating their differences. At the next iteration of the DGS algorithm, conflicted two-exchanges may be resolved and applied in the parallel section of the algorithm. In order to execute the parallel Switching phase on the GPU, we simply let the GPU spawn a number of threads which is equal to the number of non-conflicting 2-exchanges and let them perform the switch. This modification not only implies the Switching phase to be executed on the GPU, but it makes also possible for the solver to be run completely on it. As a direct consequence, the number of memory transfers between host and device are reduced dramatically. In fact, now only the input and output of the solver are transferred from/to the GPU.

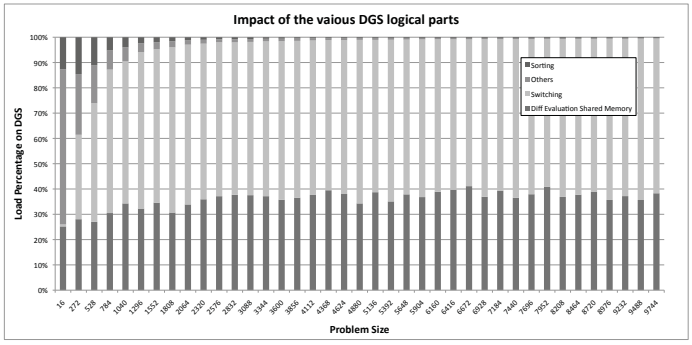

Fig. 2. Relative computational load of the various parts of the DGS solver.

VI. General Results

In Figure 3 we show the results obtained by comparing three different implementations of the DGS heuristic: a pure C implementation labeled "All CPU DGS", the "DGS Mixed DGS" implementation, where only the Difference Evaluation and the Sorting sections of the algorithm are executed on the GPU using Shared Memory, and the "GPU DGS" implementation, where all three main parts of the DGS including the Switching are executed on the GPU. As we can observe, the gain in performance when considering the "GPU DGS" compared to the two other implementations is paramount. There are two fundamental reasons for that. The first is the speed-up obtained by applying all non-conflicting 2-exchanges in parallel.

The second reason is a direct consequence of the fact that most of the operations are executed directly on the GPU and few host–device operations are needed. Such operations, e.g. memory transfers, can be expensive and certainly contribute to the absolute time needed for the solver to reach an outcome. In fact, it's interesting to observe that the total termination time

ALGORITHM DGS (σ, f)
repeat
  $\sigma_{start} \leftarrow \sigma$, $\tau = M(\sigma)$, $\overline{\delta} \leftarrow \emptyset$, $\underline{\delta} \leftarrow \emptyset$
  start parallel $\forall i \in I, \forall j \in J$ ▷ Difference Evaluation Phase starts
  $ADE(i, f, \tau, \sigma, NA, \overline{\delta})$
  $JDE(j, f, \tau, \sigma, NJ, \underline{\delta})$
  stop parallel ▷ Difference Evaluation Phase ends
  while $\exists \overline{\delta}_i > 0 \vee \exists \underline{\delta}_j > 0$ **do** ▷ Switching phase
    $\boldsymbol{CRC}(I, J, NA, NJ, C)$ $\delta_i \leftarrow 0 \quad \forall i \in I$,
    $\delta_{n+j} \leftarrow 0 \quad \forall j \in J$
    $\delta_i \leftarrow \{\overline{\delta}_i \mid i \notin C\} \quad \forall i \in I$
    $\delta_{n+j} \leftarrow \{\underline{\delta}_j \mid \sigma(j) \notin C\} \quad \forall j \in J$
    start parallel $\forall \delta_t > 0$
    if $t \leq n$ **then**
      $i \leftarrow t$, $\sigma' \leftarrow$ **SWITCH**(i, j_i, σ)
    else
      $j \leftarrow (t - n)$, $\sigma' \leftarrow$ **SWITCH**(i_j, j, σ)
    if $f(\sigma') > f(\sigma)$ **then**
      $\sigma \leftarrow \sigma'$, $\tau = M(\sigma)$
    stop parallel
    start parallel
    $\forall i \in \{I \mid i \notin C\}, \forall j \in \{J \mid \sigma(j) \notin C\}$
    $ADE(i, f, \tau, \sigma, NA, \overline{\delta})$
    $JDE(j, f, \tau, \sigma, NJ, \underline{\delta})$
    stop parallel
until $f(\sigma_{start}) = f(\sigma')$
output σ'

Algorithm 3: Parallel DGS

ALGORITHM CC (I, NA, C)
$CR \leftarrow \emptyset$, $C \leftarrow \emptyset$
foreach $i \in \{I \mid NA_i \neq 0\}$ **do**
  $\sigma \leftarrow NA_i$
  $i' \leftarrow \sigma(j_i)$
  if $i \in CR$ **or** $i' \in CR$ **then**
    $C \leftarrow \{C, i\}$
  else
    $CR \leftarrow \{CR, i, i'\}$
foreach $j \in \{J \mid NJ_j \neq 0\}$ **do**
  $\sigma \leftarrow NJ_j$
  $i \leftarrow \sigma(j)$
  if $i \in CR$ **or** $i_j \in CR$ **then**
    $C \leftarrow \{C, i\}$
  else
    $CR \leftarrow \{CR, i, i_j\}$

Algorithm 4: Check Conflicts

needed for big problem size is less than the total time needed for executing just the ADE/JDE phase, as shown in Figure 1, where multiple memory transfers occur at every iteration of the algorithm.

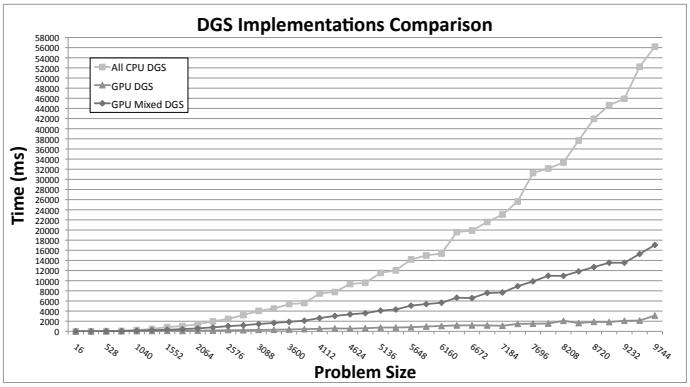

Fig. 3. Computational time comparison between DGS's implementations.

VII. Conclusion & Future Work

In this paper we presented the realization of a GPU-enabled solver based on the Deep Greedy Switching heuristic algorithm and implemented using the CUDA programming language. We detailed the process of implementation and enhancement of the two main parts of the algorithm: Difference Evaluation and Switching, and we provided results showing the impact of each iteration on the performance of the solver. In particular, we showed how parallelizing some parts of the solver with CUDA can lead to substantial speed–ups. We also suggested a modification to the DGS algorithm, in the Switching section, which enables the solver to be executed totally on GPU. In the last part of the paper, we also show the performance of the final version of the solver compared to a pure C language DGS implementation and to an auction algorithm implementation on GPUs, concluding that the time needed for the DGS solver to reach an outcome is one order of magnitude lower compared to the "C" implementation for big scenarios.

For future work, we would like to formally analyze the modified version of the DGS algorithm to theoretically assess its lower bound on optimality. We would also like to see our solver applied in different contexts and explore possible applications involving LSAP that have yet to be investigated due to computational limitations.

References

[1] A. Naiem and M. El-Beltagy, "Deep greedy switching: A fast and simple approach for linear assignment problems," in *7th International Conference of Numerical Analysis and Applied Mathematics*, 2009.

[2] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with cuda," *Micro, IEEE*, vol. 28, no. 4, pp. 13–27, 2008. [Online]. Available: http://dx.doi.org/10.1109/MM.2008.57

[3] K. Group. Opencl: The open standard for parallel programming of heterogeneous systems. [Online]. Available: http://www.khronos.org/opencl/

[4] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *Micro, IEEE*, vol. 28, no. 2, pp. 39–55, May 2008. [Online]. Available: http://dx.doi.org/10.1109/MM.2008.31

[5] A. Bayoumi, M. Chu, Y. Hanafy, P. Harrell, and G. Refai-Ahmed, "Scientific and engineering computing using ati stream technology," *Computing in Science and Engineering*, vol. 11, pp. 92–97, 2009.

[6] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, pp. 83–97, 1955.

[7] D. Bertsekas, "The auction algorithm: A distributed relaxation method for the assignment problem," *Annals of Operations Research*, vol. 14, no. 1, pp. 105–123, 1988.

[8] L. Bus and P. Tvrdık, "Distributed Memory Auction Algorithms for the Linear Assignment Problem," in *Proceedings of 14th IASTED International Conference of Parallel and Distributed Computing and Systems*, 2002, pp. 137–142.